Practical Work Part 3: Direct manipulation

This section is dedicated to the development of a rich graphical interface with direct manipulation of graphical shapes. **duration: 4h**

<u>Objectives:</u>

- draw geometric shapes using high-level JavaFX classes allowing interactions,
- manipulate the scene graph by applying transformations to its nodes,
- implement direct manipulation interactions such as pan, drag and mouse centered zoom.

<u>Documentation:</u>

JavaFX Layouts
http://docs.oracle.com/javase/8/javafx/layout-tutorial/index.html

JavaFX Events processing
http://docs.oracle.com/javase/8/javafx/events-tutorial/processing.htm

Applying Transformations in *JavaFX*
http://docs.oracle.com/javase/8/javafx/visual-effects-tutorial/transforms.htm#CHDGCBAH

JavaFX Javadoc
https://docs.oracle.com/javase/8/javafx/api/toc.htm

<u>Exercise 1: Graphic shapes</u>

1.1 Launch *NetBeans* and create a new project named “PracticalWork3” without main class (uncheck the box “Create Application Class”). Copy then Paste (Paste > Copy contextual menu) the code and resources provided with this subject on e-campus.

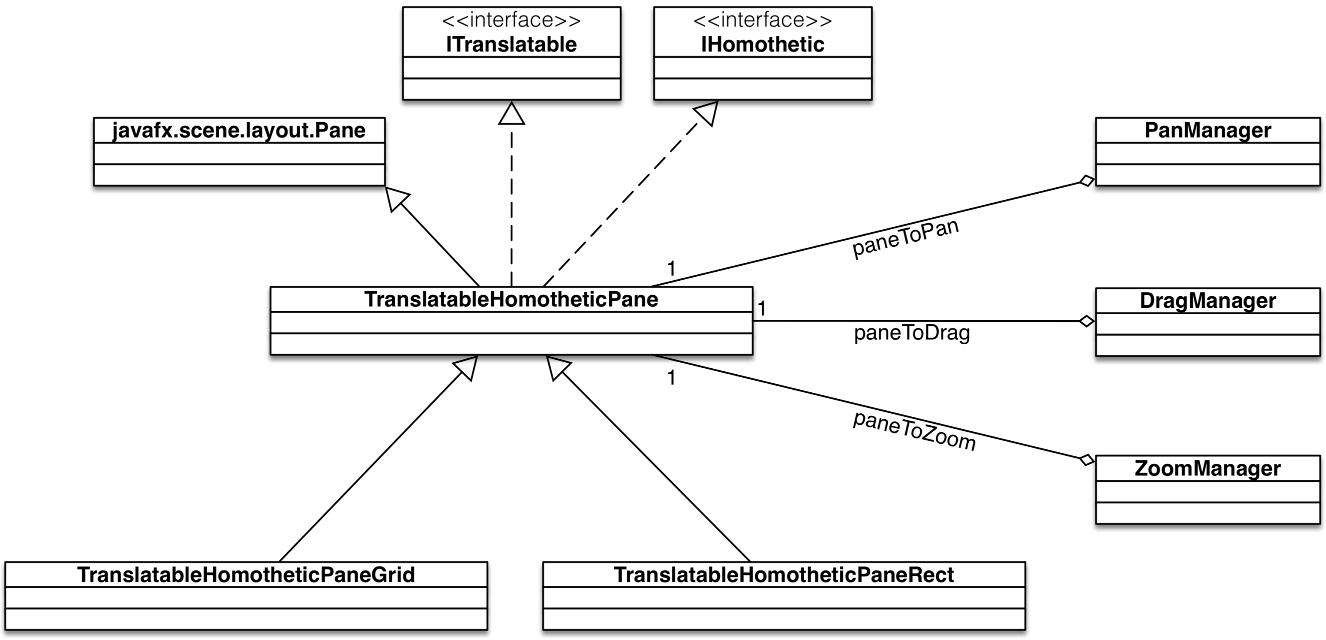
This code will be used as a basis for creating direct manipulation interactions of one or more graphical shapes in a container that can itself be manipulated. The manipulations will be as follows:

- pan of the container (translation of the container, and thus of everything it contains, by direct manipulation with the mouse),
- drag of the shapes (translation independently of the container by direct manipulation with the mouse),
- zoom (of the container) centered on the position pointed by the mouse.

The application will have the following general specifications:

- the management of interactions shall be implemented in the `PanManager`, `DragManager` and `ZoomManager` classes,
- the container shall display a grid in the background and shall be implemented as a class named `TranslatableHomotheticPaneGrid`,
- the graphical shape to manipulate shall be a rectangle and shall be implemented as a class named `TranslatableHomotheticPaneRect`.

Analyze the class diagram extract below showing schematically the classes provided (which will be completed during the work session) and a few others (which have yet to be created). Explain the choices that have been made for the architecture of the application.



1.2 The rest of the exercise is devoted to drawing the grid in the `TranslatableHomotheticPaneGrid` class. To prepare this work, browse the javadoc of the package `javafx.scene.shape` and choose the class(es) of graphic objects that can be used to draw the grid.

Knowing that you will have to integrate the grid in a `Group` (another type of *JavaFX* container), itself integrated in the scene graph of the class, draw the complete scene graph of the application.

1.3 With the preparatory work for question 1.2 completed, now code the `TranslatableHomotheticPaneGrid` class.

In the remainder of this work session, to avoid heaviness in the statements, each mention to the grid will evoke the instance of the class containing the grid (`TranslatableHomotheticPaneGrid`) and no longer the lines that make up the grid itself.

<u>Exercise 2: Pan the grid</u>

2.1 Positioning and translation

Each node of the scene graph is positioned in its parent's coordinate system thanks to two mechanisms that do more or less the same thing (placing the node in x,y) but are offered for convenience: one is supposed to deal with a supposed absolute positioning (`layoutX` and `layoutY` attributes), and the other with translations that one would like to do additionally (`translateX` and `translateY` attributes).

In `DragPanZoomApplication`, the mechanism associated with the layout is used to position the different nodes when they are created. In the following exercises you will use instead the translation mechanism provided by the transformation matrices.

From the course identify the methods associated with these mechanisms and how they are used in the code provided (and should continue to be used for the interactions to be implemented).

2.2 Using the pan logic (described below) and the `MouseEvent` doc, determine the different types of this event and the conveyed information potentially useful for managing the pan.

Pan logic:

- when the mouse button is pressed, its P_0 position must be stored,
- When the mouse is moved to the point P_1, the object being listened to must be translated by a vector P_0P_1,
- if the coordinates are in the translated object's coordinate system there is nothing else to do, if they are in its parent's coordinate system then the coordinates of P_1 must be stored in P_0 to serve as a new basis for the next mouse movement.

2.3 In the package `javafxdragpanzoom.view.control` create the class `PanManager` in charge of the pan of the grid. You will use the full subscription by adding the `EventHandler<MouseEvent>` as a filter.

2.4 Open the `DragPanZoomApplication` class and integrate an instance of your new `PanManager` class associated with your grid. Test your application. What happens?

Hint: the provided `TranslatableHomotheticPane` class is actually only partially implemented (enough for the first version of the application to work in Exercise 1). In this skeleton, all the methods specified by the `ITranslatable` interface had to be implemented if the application was to be usable. These methods could have been implemented empty but the choice that has been made here is rather to propagate an exception. Is this exception under compiler control or not? Explain the reason for this.

2.5 Take again the code of `TranslatableHomotheticPane` and implement there the function necessary for the operation of the pan, using the matrix operations seen during the course.

<u>Exercise 3: Drag the rectangle</u>

3.1 We now wish to move the rectangle independently of the grid in which it is located.

Considering that pan and drag are similar (translation of a node of the graph by direct manipulation), put it in a new `DragManager` class (still in the `javafxdragpanzoom.view.control` package).

3.2 Test your drag. There seems to be a conflict between several interactions. Where does it come from and how do you deal with it? Describe two distinct solutions (think about the properties carried by an event for a solution and propagation cycle of events discussed in previous work sessions for the other solution).

3.3 Implement one of the two solutions at your convenience.

<u>Exercise 4: Basic Zoom</u>

4.1 Following the same logic as what was done for the `translate(double dx, double dy)` method in questions 2.4 and 2.5 implement the `appendScale(double scale)` method (carefully read the javadoc of `IHomothetic` interface and the comments of `TranslatableHomotheticPane` class).

4.2 Test the result by modifying the keyboard `EventHandler` in `DragPanZoomApplication` to zoom in when a key of your choice is pressed. What is the center of the transformation thus achieved?

4.3 Using the `ScrollEvent` documentation, determine the type of this event and the information conveyed that may be useful to manage a zoom of the grid by action on the mouse wheel.

4.4 Create the `ZoomManager` class to manage the zoom of the grid (as for other interaction managers, place it in the `javafxdragpanzoom.view.control` package).

With always the same centre of transformation is this a practical interaction? What might we prefer to have?

Exercise 5: Mouse centered zoom

5.1 Now implement the `appendScale(double scale, double x, double y)` method for scaling around any point and test it by choosing an arbitrary point that is easy to spot (one of the corners of the grid for example).

5.2 Use the `ScrollEvent` documentation to determine the additional information to be used to manage a mouse centered zoom.

5.3 Improve the zoom implemented in Exercise 4.

Exercise 6: Differentiated zoom

6.1 Using the previously made zoom causes what viewing problem, especially at very small or very large zoom levels?

What solution can be envisaged? Hint: Take a look at how zoom is handled in google maps.

6.2 What type of application can there be for your project?

6.3 We are going to set up a first so-called differentiated zoom, which will be applied to the grid. It will ensure that the thickness of its lines remains visually invariant as the scale changes.

Be careful, all the transformations being cumulated along the scene graph, visually invariant does not really mean invariant. Rather, it means that as you apply transformations to the grid, you will have to apply the inverse transformations to the nodes inside the grid whose aspect you do not want to change (or to a particular property of these nodes so that it remains visually invariant after the accumulation of all the transformations).

In our case, the processing wont be done on the grid as a whole or even on the lines that compose it (the `Line` instances) but on the `strokeWidth` property of each of these lines, which will have to be divided by the value of the scale (the `scale` property inherited from `TranslatableHomotheticPane`) each time it changes.

In order to be able to react to changes in the value of the scale property, a `ChangeListener` should be used, as in the very first work session.

6.4 To what type of visual component of your project will you have to apply the implemented mechanism?

6.5 Now let's move on to the rectangle. It requires a specific differentiated zoom mechanism: its size should not change at all on the screen when zooming in. This time the transformation is simpler since it is global: when the scale property of the grid changes we just have to apply the inverse scale to the whole rectangle. To do so we need:

- to pass a reference to this `scale` property from the grid class to the `TranslatableHomotheticPaneRect` class,
- to implement the `setScale(double scale)` method,
- as far as we are there, let's also implement the `setScale(double scale, double pivotX, double pivotY)` method to have a complete and consistent API (don't need it right here but think reusability!).

Congratulations! At the end of this working session you know enough to develop standard direct manipulation interactions. For your project you will need a few more notions. We will address them during dedicated sessions and according to your progress. Begin to think about it, it's up to you now!

[Exercise 7: Alternative to inverse transformations]

We made our differentiated zooms by applying inverse transformations. Isn't there another way to do differentiated zooming? Think about a different organization of the scene graph and the use of bindings.

What would be the advantages and disadvantages compared to the chosen solution?